6.096 Problem Set 3

1 Additional Material

1.1 Arrays of class objects

An array of class objects is similar to an array of some other data type. To create an array of `Points`, we write

```
Point parray[4];
```

To access the object at position i of the array, we write

```
parray[i]
```

and to call a method on that object method, we write

```
parray[i].methodName(arg1, arg2, ...);
```

To initialize an array of objects whose values are known at compile time, we can write

```
Point parray[4] = {Point(0,1), Point(1,2), Point(3,5), Point(8,13)};
```

We can also allocate an array of objects dynamically using the `new` operator (this implicitly calls the default constructor of each new `Point`):

```
Point* parray = new Point[4];
```

1.2 Static members and variables

Static data members of a class are also known as "class variables," because there is only one unique value for all the objects of that class. Their content is not different from one object of this class to another.

For example, it may be used for a variable within a class that can contain a counter with the number of objects of the class that are currently allocated, as in the following example:

```
#include <iostream>

using namespace std;

class CDummy
{
```

```
7 public:
8     static int n;
9     CDummy() { ++n; }
10    ~CDummy() { --n; }
11 };
12
13 int CDummy::n = 0;
14
15 int main()
16 {
17     CDummy a;
18     CDummy b[5];
19     CDummy* c = new CDummy;
20     cout << a.n << "\n"; // prints out 7
21     delete c;
22     cout << CDummy::n << "\n"; // prints out 6
23     return 0;
24 }
```

In fact, static members have the same properties as global variables, but they can only be referenced via the class: either in class methods, via a class instance (`someObject.staticVariable`, or via the `className::variable` construct.

Because these variables are global, if we were to initialize them in a header file we could end up with that initialization being compiled multiple times (once per time we include the header). To avoid this, we only include a static member's "prototype" (its declaration) in the class declaration, but not its definition (its initialization). This is why line 13 above is necessary, and why if we were to provide a header file for `CDummy`, we would still need to put line 13 in a separate `.cpp` file. If you get linker errors saying a `static int` is undefined, check to see whether you've included a line like line 13 in a `.cpp` file.

Classes can also have `static` member functions – that is, member functions which are associated with the class but do not operate on a particular class instance. Such member functions may not access non-`static` data members. For instance, we might replace `CDummy` above with the following class definition:

```
class CDummy
{
private:
    static int n;
public:
    CDummy() { ++n; }
    ~CDummy() { --n; }
    static int getN() {return n;}
};
```

`getN` could then be called as `c->getN()` or `CDummy::getN()`.

1.3 `const` member functions

It is clear what `const`-ness means for a simple value like an `int`, but it is not clear what functions should be available on a `const` object, since functions may allow modifications in subtle ways that ought to be forbidden on `const` objects. To specify to the compiler that a given member function is safe to call on `const` objects, you can declare the function with the `const` keyword. This specifies that the function is a "read-only" function that does not modify the object on which it is called.

To declare a `const` member function, place the `const` keyword after the closing parenthesis of the argument list. The `const` keyword is required in both the prototype and the definition. A `const` member function cannot modify any data members or call any member functions that aren't also declared `const`. Generally, `const` member functions should return `const` values, since they often return references/pointers to internal data, and we wouldn't want to allow someone to get a modifiable reference to the data of a `const` object.

```cpp
const string &Person::getName() const {
    return name;    // Doesn't modify anything; trying to modify a
                    // data member from here would be a syntax error
}
```

If an object of class `Person` would be declared as `const Person jesse;`, no non-`const` member functions could be called on it. In other words, the set of `const` member functions of a class defines the set of operations that can be performed on `const` objects of that class.

1.4 String objects

Manipulating strings as `char *` or `char[]` types tends to be unwieldy. In particular, it is difficult to perform modifications on strings that change their length; this requires reallocating the entire array. It is also very difficult to deal with strings whose maximum length is not known ahead of time. Many C++ classes have been created to solve such problems. The C++ standard library includes one such class, appropriately called `string`, defined in header file `string` under namespace `std`.

The `string` class allows us to do all sorts of nifty operations:

```cpp
#include <string>
...
string s = "Hello";
s += " world!";
if(s == "Hello world!") {
    cout << "Success!" << endl;
}
cout << s.substr(6, 6) << endl; // Prints "world!"
cout << s.find("world"); // (prints "6")
cout << s.find('l', 5); // (prints "9")
```

A line-by-line description of the `string` features this code demonstrates:

3. We can set strings to normal `char *`'s.

4. We can use the `+` operator to append things to a string. Don't worry about how this works for now; we'll see in Lecture 9 how to allow your classes to do things like this.)

5. We can use the `==` operator to test whether two strings are the same. (If we tried to do this with `char *`'s, we'd just be checking whether they point to the same string in memory, not whether the pointed-to strings have the same contents. To check for string equality with `char *`'s, you need to use the function `strcmp`.)

8. We can get a new `string` object that is a substring of the old one.

9. We can find the index a given string within the `string` object.

10. We can find a character as well, and we can specify a starting location for the search.

Take a few minutes to play around with the string class. Look at the documentation at http://www.cplusplus.com/reference/string/string/. In particular, be sure to understand the behavior of the `substr` function.

1.5 Type Conversions and Constructors

Any time you call a function, the compiler will do its best to match the arguments you provide with some function definition. As a last-ditch strategy, it will even try constructing objects for you.

Say you have a function `f` that takes a `Coordinate` object, and that the `Coordinate` constructor is defined to take one `double`. If you call `f(3.4)`, the compiler will notice that there is no `f` that takes a `double`; however, it will also see that it can match the `f` that it found by converting your argument to a `Coordinate` object. Thus, it will automatically turn your statement into `f(Coordinate(3.4))`.

This applies to constructors, as well. Say you have a `Point` class, whose constructor takes two `Coordinate`s. If you write `Point p(2.3, 0.5);`, the compiler will automatically turn your statement into `Point p(Coordinate(2.3), Coordinate(2.5);`.

1.6 Sources

- http://www.cplusplus.com/

2 Catch that bug

In this section, the following snippets will have bugs. Identify them and indicate how to correct them. Do these without the use of a computer!

2.1

```cpp
...
class Point
{
private:
    int x, y;

public:
    Point(int u, int v) : x(u), y(v) {}
    int getX() { return x; }
    int getY() { return y; }
    void doubleVal()
    {
        x *= 2;
        y *= 2;
    }
};

int main()
{
    const Point myPoint(5, 3)
    myPoint.doubleVal();
    cout << myPoint.getX() << " " << myPoint.getY() << "\n";
    return 0;
}
```

2.2

```cpp
...
class Point
{
private:
    int x, y;

public:
    Point(int u, int v) : x(u), y(v) {}
    int getX() { return x; }
    int getY() { return y; }
    void setX(int newX) const { x = newX; }
};

int main()
{
    Point p(5, 3);
    p.setX(9001);
        cout << p.getX() << ' ' << p.getY();
    return 0;
}
```

2.3

```cpp
...
class Point
{
private:
    int x, y;

public:
    Point(int u, int v) : x(u), y(v) {}
    int getX() { return x; }
    int getY() { return y; }
};

int main()
{
    Point p(5, 3);
    cout << p.x << " " << p.y << "\n";
    return 0;
}
```

2.4

```cpp
...
class Point
{
private:
    int x, y;

public:
    Point(int u, int v) : x(u), y(v) {}
    int getX() { return x; }
    void setX(int newX);
};

void setX(int newX){ x = newX; }

int main()
{
    Point p(5, 3);
    p.setX(0);
    cout << p.getX() << " " << "\n";
    return 0;
}
```

2.5

```
...
int size;
cin >> size;
int *nums = new int[size];
for(int i = 0; i < size; ++i)
{
    cin >> nums[i];
}
... // Calculations with nums omitted
delete nums;
...
```

2.6

```
class Point
{
private:
    int x, y;

public:
    Point(int u, int v) : x(u), y(v) {}
    int getX() { return x; }
    int getY() { return y; }
};

int main()
{
    Point *p = new Point(5, 3);
```

```
        cout << p->getX() << ' ' << p->getY();
    return 0;
}
```

(Hint: this bug is a logic error, not a syntax error.)

3 Point

For the next several problems, you should put your class definitions and function prototypes in a header file called `geometry.h`, and your function definitions in a file called `geometry.cpp`. If your functions are one-liners, you may choose to include them in the header file.

In this section you will implement a class representing a point, appropriately named `Point`.

3.1 Foundation

Create the class with two private `int`s. Name them `x` and `y`.

3.2 Constructors

Implement a single constructor that, if called with 0 arguments, initializes a point to the origin – $(0, 0)$ – but if called with two arguments x and y, creates a point located at (x, y). (*Hint:* You will need to use default arguments.

3.3 Member Functions

Support the following operations using the given function signatures:

- Get the x coordinate

  `int Point::getX() const`

- Get the y coordinate

  `int Point::getY() const`

- Set the x coordinate

  `void Point::setX(const int new_x)`

- Set the y coordinate

  `void Point::setY(const int new_y)`

4 `PointArray`

In this section you will implement a class representing an array of `Point`s. It will allow dynamically resizing the array, and it will track its own length so that if you were to pass it to a function, you would not need to pass its length separately.

4.1 Foundation

Create the class with two private members, a pointer to the start of an array of `Point`s and an `int` that stores the size (length) of the array.

4.2 Constructors

Implement the default constructor (a constructor with no arguments) with the following signature. It should create an array with size 0.

Implement a constructor that takes a `Point` array called `points` and an `int` called `size` as its arguments. It should initialize a `PointArray` with the specified size, copying the values from `points`. You will need to dynamically allocate the `PointArray`'s internal array to the specified size.

```
PointArray::PointArray(const Point points[], const int size)
```

Finally, implement a constructor that creates a copy of a given `PointArray` (a *copy constructor*).

```
PointArray::PointArray(const PointArray& pv)
```

(*Hint*: Make sure that the two `PointArray`s do not end up using the same memory for their internal arrays. Also make sure that the contents of the original array are copied, as well.)

4.3 Destructors

Define a destructor that deletes the internal array of the `PointArray`.

```
PointArray::~PointArray()
```

4.4 Dealing with an ever-changing array

Since we will allow modifications to our array, you'll find that the internal array grows and shrinks quite often. A simple (though very inefficient) way to deal with this without repetitively writing similar code is to write a member function `PointArray::resize(int n)` that allocates a new array of size n, copies the first $\min(\text{previous array size}, n)$ existing elements into it, and deallocates the old array. If doing so has increased the size, it's fine for `resize` to leave the new spaces uninitialized; whatever member function calls it will be responsible for filling those spaces in. Then every time the array size changes at all (including `clear`), you can call this function.

In some cases, after you call this function, you will have to subsequently shift some of the contents of the array right or left in order to make room for a new value or get rid of an old one. This is of course inefficient; for the purposes of this exercise, however, we won't be worrying about efficiency. If you wanted to do this the "right" way, you'd remember both how long your array is and how much of it is filled, and only reallocate when you reach your current limit or when how much is filled dips below some threshhold.

Add the `PointArray::resize(int n)` function as specified above to your `PointArray` class. Give it an appropriate access modifier, keeping in mind that this is meant for use only by internal functions; the public interface is specified below.

4.5 Member Functions

Implement public functions to perform the following operations:

- Add a `Point` to the end of the array

  ```
  void PointArray::push_back(const Point &p)
  ```

- Insert a `Point` at some arbitrary position (subscript) of the array, shifting the elements past `position` to the right

  ```
  void PointArray::insert(const int position, const Point &p)
  ```

- Remove the `Point` at some arbitrary position (subscript) of the array, shifting the remaining elements to the left

  ```
  void PointArray::remove(const int pos)
  ```

- Get the size of the array

  ```
  const int PointArray::getSize() const
  ```

- Remove everything from the array and sets its size to 0

  ```
  void PointArray::clear()
  ```

- Get a pointer to the element at some arbitrary position in the array, where positions start at 0 as with arrays

  ```
  Point *PointArray::get(const int position)
  ```

  ```
  const Point *PointArray::get(const int position) const
  ```

If `get` is called with an index larger than the array size, there is no `Point` you can return a pointer to, so your function should return a null pointer. Be sure your member functions all behave correctly in the case where you have a 0-length array (i.e., when your `PointArray` contains no points, such as after the default constructor is called).

4.5.1

Why do we need `const` and non-`const` versions of `get`? (Think about what would happen if we only had one or the other, in particular what would happen if we had a `const PointArray` object.)

5 Polygon

In this section you will implement a class for a convex polygon called `Polygon`. A *convex polygon* is a simple polygon whose interior is a convex set; that is, if for every pair of points within the object, every point on the straight line segment that joins them is also within the object.

`Polygon` will be an *abstract class* – that is, it will be a placeholder in the class hierarchy, but only its subclasses may be instantiated. `Polygon` will be an immutable type – that is, once you create the `Polygon`, you will not be able to change it.

Throughout this problem, remember to use the `const` modifier where appropriate.

5.1 Foundation

Create the class with two protected members: a `PointArray` and a `static int` to keep track of the number of `Polygon` instances currently in existence.

5.2 Constructors/Destructors

Implement a constructor that creates a `Polygon` from two arguments: an array of `Point`s and the length of that array. Use member initializer syntax to initialize the internal `PointArray` object of the `Polygon`, passing the `Polygon` constructor arguments to the `PointArray` constructor. You should need just one line of code in the actual constructor body.

Implement a constructor that creates a polygon using the points in an existing `PointArray` that is passed as an argument. (For the purposes of this problem, you may assume that the order of the points in the `PointArray` traces out a convex polygon.) You should make sure your constructor avoids the unnecessary work of copying the entire existing `PointArray` each time it is called.

Will the default "memberwise" copy constructor work here? Explain what happens to the `PointArray` field if we try to copy a `Polygon` and don't define our own copy constructor.

Make sure that your constructors and destructors are set up so that they correctly update the `static int` that tracks the number of `Polygon` instances.

5.3 Member Functions

Implement the following public functions according to the descriptions:

- `area`: Calculates the area of the `Polygon` as a `double`. Make this function pure virtual, so that the subclasses must define it in order to be instantiated. (This makes the class abstract.)
- `getNumPolygons`: Returns the number of `Polygons` currently in existence, and can be called even without referencing a `Polygon` instance. (*Hint:* Use the `static int`.)
- `getNumSides`: Returns the number of sides of the `Polygon`.
- `getPoints`: Returns an unmodifiable pointer to the `PointArray` of the `Polygon`.

5.4 Rectangle

Write a subclass of `Polygon` called `Rectangle` that models a rectangle. Your code should

- Allow constructing a `Rectangle` from two `Points` (the lower left coordinate and the upper right coordinate)
- Allow construct a `Rectangle` from four `ints`
- Override the `Polygon::area`'s behavior such that the rectangle's area is calculated by multiplying its length by its width, but still return the area as a double.

Both of your constructors should use member initializer syntax to call the base-class constructor, and should have nothing else in their bodies. C++ unfortunately does not allow us to define arrays on the fly to pass to base-class constructors. To allow using member initializer syntax, we can implement a little trick where we have a global array that we update each time we want to make a new array of `Points` for constructing a `Polygon`. You may include the following code snippet in your `geometry.cpp` file:

```cpp
Point constructorPoints[4];

Point *updateConstructorPoints(const Point &p1, const Point &p2,
    const Point &p3, const Point &p4 = Point(0,0)) {
    constructorPoints[0] = p1;
    constructorPoints[1] = p2;
    constructorPoints[2] = p3;
    constructorPoints[3] = p4;
    return constructorPoints;
}
```

You can then pass the return value of `updateConstructorPoints(...)` (you'll need to fill in the arguments) as the `Point` array argument of the `Polygon` constructor. (Remember, the name of an array of `T`s is just a `T` pointer.)

5.5 Triangle

Write a subclass of `Polygon` called `Triangle` that models a triangle. Your code should

- Construct a `Triangle` from three `Points`
- Override the area function such that it calculates the area using Heron's formula:

$$K = \sqrt{s(s-a)(s-b)(s-c)}$$

  where a, b, and c are the side lengths of the triangle and $s = \frac{a+b+c}{2}$.

Use the same trick as above for calling the appropriate base-class constructor. You should not need to include any code in the actual function body.

5.6 Questions

1. In the `Point` class, what would happen if the constructors were private?
2. Describe what happens to the fields of a `Polygon` object when the object is destroyed.
3. Why did we need to make the fields of `Polygon` protected?

For the next question, assume you are writing a function that takes as an argument a `Polygon *` called `polyPtr`.

4. Imagine that we had overridden `getNumSides` in each of `Rectangle` and `Triangle`. Which version of the function would be called if we wrote `polyPtr->getNumSides()`? Why?

5.7 Putting it All Together

Write a small function with signature `void printAttributes(Polygon *)` that prints the area of the polygon and prints the (x, y) coordinates of all of its points.

Finally, write a small program (a `main` function) that does the following:

- Prompts the user for the lower-left and upper-right positions of a `Rectangle` and creates a `Rectangle` object accordingly
- Prompts the user for the point positions of a `Triangle` and creates a `Triangle` object accordingly
- Calls `printAttributes` on the `Rectangle` and `Triangle`, printing an appropriate message first.

6 Strings

In this section you will write a program that turns a given English word into Pig Latin. Pig Latin is a language game of alterations played in English. To form the Pig Latin version of an English word, the onset of the first consonant is transposed to the end of the word an an *ay* is affixed. Here are the rules:

1. In words that begin with consonant sounds, the initial consonant (if the word starts with ‘q’, then treat ‘qu’ as the initial consonant) is moved to the end of the word, and an “ay” is added. For example:
   - beast : east-bay
   - dough : ough-day
   - happy : appy-hay
   - question : estion-quay
2. In words that begin with vowel sounds, the syllable “way” is simply added to the end of the word.

Write a function `pigLatinify` that takes a `string` object as an argument. (You may assume that this string contains a single lowercase word.) It should return a new `string` containing the Pig Latin version of the original. (Yes, it is inefficient to copy a whole string in the return statement, but we won't worry about that. Also, your compiler is probably clever enough to do some optimizations.) You may find it useful to define a constant of type `string` or `char*` called `VOWELS`.

Remember that `string` objects allow the use of operators such as `+=` and `+`.

(Your answers for this problem should go in a separate file from `geometry.h` and `geometry.cpp`.)